Real Time Video Stitching

An Honors Thesis (HONR 499)

by

Kelly Blair

Thesis Advisor
Shaoen Wu

Signed

Ball State University

Muncie Indiana

May 2016

Expected Date of Graduation
May 2016

ABSTRACT:

The purpose of this project is to improve the speed of existing stitching algorithms in relation to video stitching. The speed can be increased on the axis of distance by focusing only on the region of image overlap. The speed can be increased on the axis of time by reusing homography information across multiple frames. It may also be possible to combine these approaches to further speed up the stitching process. The result of the project was an image stitcher that worked at an increased speed of 6.5FPS as opposed to 1.5FPS in other approaches. The downside of this approach is a drop in stitching accuracy in relation to other stitching approaches. More work is needed to fully develop a real time video stitcher.

ACKNOWLEDGEMENTS:

I would like to thank Dr. Shaoen Wu for mentoring me during this project and providing the initial theory to get me started. I would also like to thank the National Science Foundation for funding/sponsoring my research.

Real Time Video Stitching

Kelly Blair
Ball State University

1. INTRODUCTION

Image stitching is the process of combining multiple images to create a panorama. Static image stitching is a process that has been universally solved.**[4]** Although there will always be a faster and better way to do things, image stitching is available in a wide variety of commercial applications.**[15][16]**

However, video stitching is a newer field. Although there have been a few methods proposed that would work in specific situations **[17][18][19]**, no one has yet proposed a universal solution.

1.1 What is Image Stitching?

Image stitching is the process of combining multiple images to create a panorama. This is done in a series of steps called the image stitching pipeline. **[5]**

To begin, the program searches for distinct feature points in the image. For example, a white pixel surrounded by black pixels is highly unique, but a cluster of green pixels is not. **[10]** Unusual points and the points surrounding them are are stored as a feature descriptor.

If there are similar feature points in each image, they will matched together. Feature descriptors are compared against each other with a forgiveness margin. Feature points without a match within this margin are discarded. Flann based matching is thought to be the fastest currently available matching algorithm. **[12]**

Based on the locations of the matched points, the program is then able to calculate a 3D model of the area. This 3D model is referred to as the image homography. **[13]**

The images can then be warped and positioned with respect to the 3D model. **[14].** This process involves modifying the image matrices. OpenCV fully automates this process.

Sometimes there is a visible seam present after combining the images. Image blending smooths this seam so it is not as noticeable. Image feathering, or setting a gradient of transparencies, is one common approach **[11].** This "feathers" the transparencies of the images close to the seam, so there is a small region of overlap which the alpha of each image is set close to 0.5. The alpha

values approach 1 as they become closer to the original image. The alphas are usually set to one within a small area, so only the region surrounding the overlap is blended. **[21]**

1.2 Why is Video Stitching Difficult?

Although there are many algorithms available for traditional image stitching**[2][3]**, the same is not true of video stitching. This is because time is not a constraint when stitching a static images. However to stitch a video feed in real time, the calculation must be done fast enough to produce an attractive frame rate. (15+ FPS) Currently, the time to stitch two static images using the openCV library is between 1 and 3 seconds. (0.33 FPS -1 FPS). In order to achieve real time video stitching, the speed must be dramatically improved.

1.3 Related Work

There is currently commercially available software for 360 panoramic video stitching. **[17]** However, this stitching is not done in real time and only includes a small sampling of frames. The output is not a video, but a scrollable panorama. Because it does not produce a video, it has limited relevance to the work.

There is at least one example of real time video stitching software given the cameras themselves do not move. **[18][19]** I was able to find a paper for one of these approaches, and found that this was accomplished by using low resolution (240x180) and multithreading.

Finally, I was able to find one example of a video stitcher that was both real time and included movement. However, I was unable to locate the creator's techniques. Additionally, the stitch was not always immediately accurate. **[20]**

2. OUTLINING APPROACHES

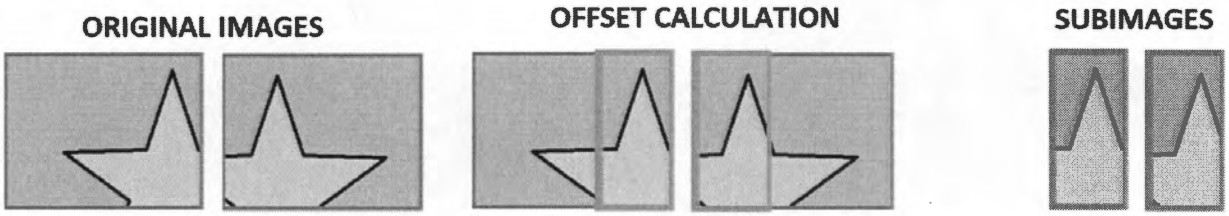
I took two approaches to computational load of the stitching calculations. The first approach involved reducing the amount of pixels needed for the homography calculation. Because this calculation takes up the majority of the stitching time, improving the speed of this calculation will also substantially improve the overall speed of the stitch.

The second approach involved reusing information calculated in previous frames. Because individual frames did not change very much, it seemed likely that information taken from a previous frame could be reused for a short period of time.

Lastly, I hoped to combine the approaches to create an additional gain in speed.

3. IMPLEMENTATION 1: HOMOGRAPHY REDUCTION

There were two available approaches to reduce the number of pixels in an image set. One option was to reduce the image resolution. The more interesting approach, however, was to focus on the overlapping region of the images. Because the image stitching calculation focuses on feature points in the overlap region, removing some extraneous edges seemed unlikely to damage the stitch integrity.



[Because the subimages are nearly identical, a homography can still be calculated]

The first thing I did was transition from using the openCV library stitch() method to something I would be able to fine tune. I found some code online that separated each individual step of the stitching pipeline[1], and decided to start from there.

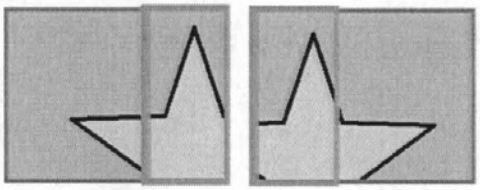
Next, I had to discover which parts of the image stitching pipeline would be different when subimages were used. I discovered that had to calculate the image homography completely from subimages. Afterwards, I needed to apply the calculated homography to the full images, and then apply an appropriate x and y translation.

The sub-imaging approach was more complicated than I anticipated. The first issue I ran into was image translation. Because the homographies were now calculated using partial images, I would need to find the correct values to translate the images to the proper position. Additionally, depending on where the two images overlapped, there may not be any one formula that could reliably find the region of overlap. For the time being, I had to manually enter these values.

Upon finding that the subimage stitch was still mostly reliable, I worked on algorithms that could automatically calculate the x and y translation, as well as the overlapping region. I also ran a series of tests to determine the ideal size of the subimage, and consistently found that values of 0.35x and 1.0y were the minimum needed to get reliable results.

To handle the x and y translation, I tentatively stored an x and y value based on the offset of the subimage from its initial position. For example, if a subimage used in homography calculation was began at an x coordinate of 100px, I would translate the warped image by 100 px. Although it was an imperfect method, it was more accurate than the default offset of zero.

OFFSET CALCULATION

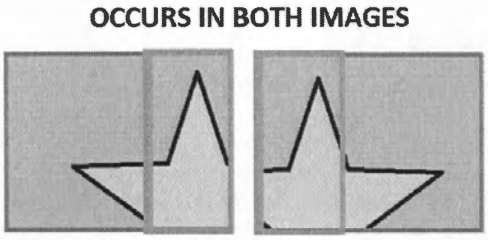


The right subimage starts at x=0, so offset = 0

Finally, I needed to find a way to programatically detect regions of overlap. This issue turned out to be the most complicated.

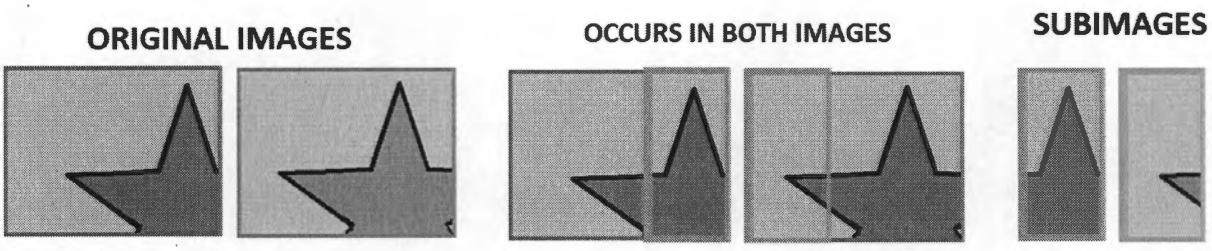
3.1 Fixed Point Subimages

My initial approach was to simply take the rightmost half of the left image, and the leftmost half of the right image. I knew that these images overlapped, so I knew that the portion of image surrounding the seam appeared in both images.



The approach worked great for a while. I tested my approach on a single set of stock photos, and it worked every time. However, once I tested some additional photos, it stopped working! I had no idea what the problem could be, since these new photos had even MORE overlap than the previous ones.

Eventually, I realized that the greater region of overlap was the problem. It was true that the subimages from each image appeared in the other image. However, the subimages themselves represented drastically different regions.



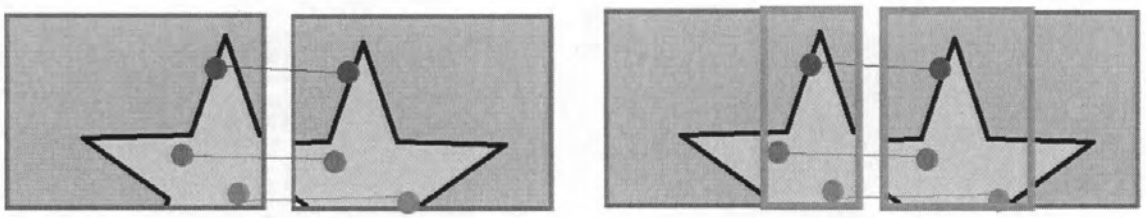
Based on the proportion of overlap, this approach had a varying degree of success. This was obviously not an acceptable solution, so I moved on to other approaches.

Set 1 **Set 2** **Set 3**

(Images from [1]

3.2 Feature Point Subimages

My next approach was to find a region of overlap based on the feature points of the two images. These feature points needed to be matched anyway as part of the stitching pipeline. Whenever two feature points are found to be a match, it implies that those points represent part of the overlapping region within each image.

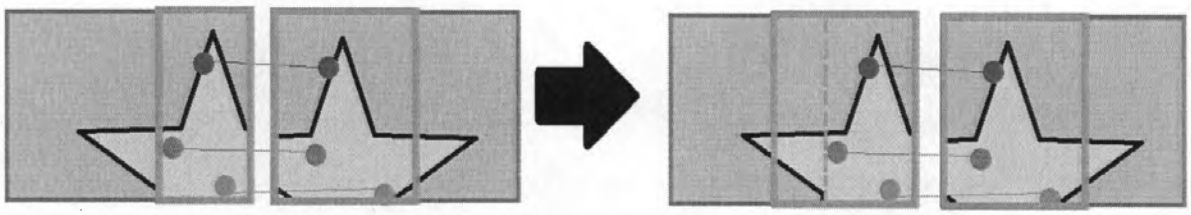


I figured I could continuously predict the subimage needed for the next frame by stitching the current one. Because this cycle was self contained, I would also occasionally run a full image stitch to recalibrate the overlap.

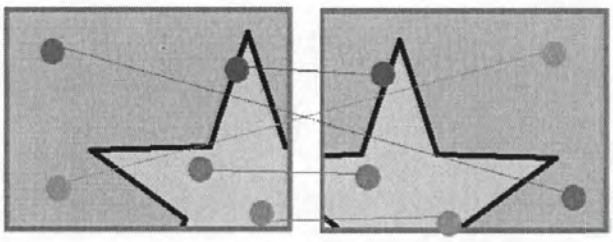
I had a few difficulties with this approach as well. The first issue was that in order to stitch images, they need to be identical in size. The subimages returned by the feature points algorithm were almost never the same size. I quickly wrote some code that chose the largest horizontal and vertical size, then applied them to both images.

The code to resize the smaller image:

1. If the images are equal in size, do nothing
2. If the smaller image was on the left, set DELTA_MULTIPLIER= -1. Else, DELTA_MULTIPLIER = 1.
3. Declare a variable X_TRANSLATION = DELTA_MULTIPLIER * (WIDTH_OF_LARGE_SUBIMAGE-WIDTH_OF_SMALL_SUBIMAGE).
4. Set X_MIN = 0.
5. Set X_MAX = ORIGINAL_IMAGE_WITDH - WIDTH_OF_LARGE_SUBIMAGE;
6. Add X_TRANSLATION to the X_COORDINATE of the smaller image.
7. If the small X_COORDINATE < X_MIN, set X_COORDINATE = X_MIN
8. If the small X_COORDINATE >X_MAX, set X-COORDINATE = X_MAX
9. Set WIDTH_OF_SMALL_SUBIMAGE = WIDTH_OF_LARGE_SUBIMAGE



Although I had solved this issue, I started to realize that even though feature point matching was used as part of image stitching, it was not 100% accurate. It seemed to be maybe 75% accurate, and it was really throwing off my sub-imaging. If I set the tolerance too high, I would get a tiny cluster of points that may or may not overlap. If I set it too low, I would get a large cluster of points spread across 80% of my images that may or may not overlap. Obviously some cleaning needed to applied to the points before they could be used.



Extraneous Points

I knew how to use standard deviation to discard outlier points, and implementing it helped a lot. I could now adjust the tolerance of the both the feature point detector and the outlier detector. But even with both of these approaches, there was still too much volatility to produce a reliable subimage. I estimate that it worked great about 50% of the time, and the other 50% it was completely useless. It was great at "fuzzy" matching, but unfortunately for this step I needed it to be fully reliable.

Sub-image Set 1

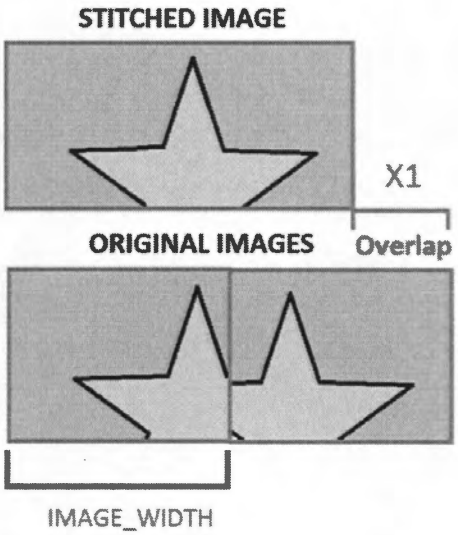
Sub-image Set 2

Sub-image Set 3

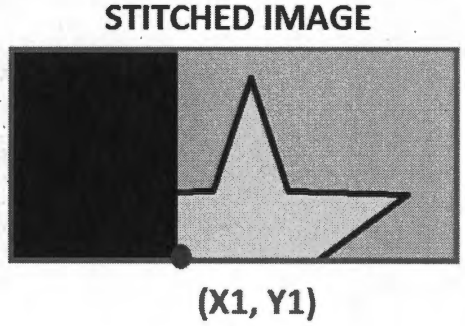
3.3 Seam Subimages

My last approach showed the most promise. I was able to repeatedly get subimages that looked almost identical. The problem was, it was incredibly challenging to implement programmatically. I called this approach "The Seam Method."

The concept was to run a single stitch using both full images, and use this information to determine the region of overlap. Based on the size of the stitched image, I could determine the region of overlap by elimination. For example, if the individual images were size 1.0x and 1.0x and the stitched image was size 1.4x, I could determine that the region of overlap in each image was close to 0.6 x. This is because if the images did not overlap at all, you the size of the stitched image would be 2.0x. Any amount less than that implies a region of overlap, which would appear equally in both images.



I stored both full images and the stitched image in memory. This stitched image is unique in that I made the left image invisible. That allowed me to see the bottom left corner of the right image. We will call the coordinates of this point X1 and Y1. We will call the original image width IMAGE_WIDTH and the width of sub-image we want to calculate ROI_WIDTH.



Because of the way openCV displays stitched images, the stitched images themselves would always be the same size. They were just superimposed on a large file with a black background. The first thing I did was write a program to trace a line around the image on this background, and remember each point on that line.

Next, the computer needed to identify the corners of each of the images. This is the part that I was not able to implement programmatically. Because the shapes were more like parallelograms than rectangles, I was not able to come up with a system which universally discovered the bottom left corner. However, I did come up with an approach for doing this manually, so that the values would be calculated the same way each time.

Finally, the corners of the image would be compared to the boundaries of the unwarped images.

This is how the final subimages were calculated.

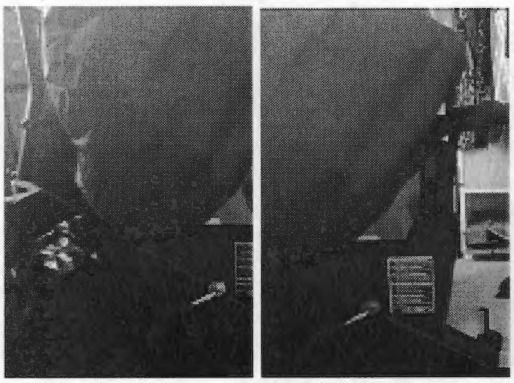
1. The right subimage is always the leftmost portion of the image. Take the original right image and calculate the subimage from X=0 to X = ROI_WIDTH
2. Start with at the left side of the original left image (X=0), then add the X1. (X=X1) This is where the left sub image starts. Calculate the subimage from X=X1 to X= X1+ROI_WIDTH
3. To get the X translation: X` = IMAGE_WIDTH - ROI_WIDTH - X1
4. To get the Y translation: Y`= -0.5Y1

Unfortunately, I was unable to come up with an accurate algorithm for finding a corner of a warped image, and this must be done manually. However, when the process is followed it almost universally produces accurate subimages. Further work could be done to automate this approach.

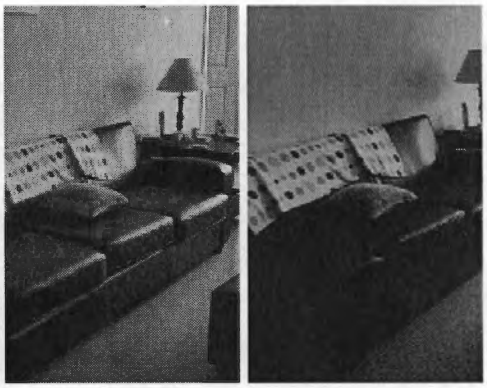
3.4 Subimage Results

The Seam Method is the most reliable approach I could come up with for calculating overlapping regions. I have shown a few examples of overlapping subimages below.

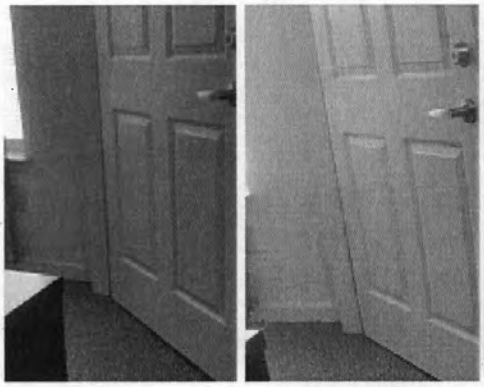
SET 1



SET 2



SET 3

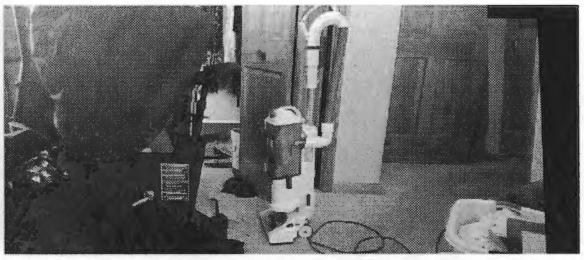


SET 4

As you can see, these sets are nearly identical. They are not perfect matches, but are as close as can reasonably be expected given the differences in the original images. As far as we know, this method for determining image overlap is unique, and the only method that exists.

3.5 Stitching Results - Visual Comparison

[FULL STITCH 1]



[SEAM STITCH 1]

[FULL STITCH 2]

[SEAM STITCH 2]

[FULL STITCH 3]

[SEAM STITCH 3]

[FULL STITCH 4]

[SEAM STITCH 4]

As you can see in image sets 1,2, and 4, the stitching is nearly identical. Although Seam Stitch 4 has alignment issues, those alignment issues are also present in Full Stitch 4 and are not caused by the Seam Imaging. However, in Seam Stitch 2 there is a large offset that is directly caused by the inaccuracy of the sub image stitching technique.

3.6 Results and Conclusion

Assuming the translation values and region of overlap are calculated properly, stitching using subimages offers an improvement in stitch time. My time trials found that using a sub image with a size of 0.35x and 1.0y of the original image offers a 20.83% increase in speed over traditional approaches. However, regardless of the method used to calculate the sub image, this method also causes a minor drop in accuracy. Using a subimage that is nonideal results in further accuracy loss.

SUBIMAGE SPEED VS FULL IMAGE SPEED

	TIME TO CALCULATE 20 STITCHES	FPS	HPS
FULL IMAGE	13.79 seconds	1.450	1.450
SUBIMAGE	11.41 seconds	1.752	2.752

PERCENT GAIN FPS = (1.752-1.450)/1.450 = 20.83%

Even a minor accuracy loss is not worth it for such a low gain. Because the speed gain was so low, and there were occasionally problems beyond a minor accuracy loss, I cannot recommend using this method as an improvement over current methods for real-time video stitching,

By using subimages, I was able to improve stitch time at the cost of image quality. It is possible to further improve this approach in the future by refining the translation algorithm to enable automation and utilizing image blending techniques to improve accuracy. If the accuracy was further improved, this approach would certainly be usable.

4. IMPLEMENTATION 2: INFORMATION REUSE

Having finished my subimage approach, I moved on to the idea of reusing information from frame to frame. My best idea was to recalculate the image homography only every X frames. This would likely improve the speed because homography calculation is a very large part of the image stitch time, and the changes from frame to frame were unlikely to warrant a homography calculation each time. I tested this tentatively with a slow moving video feed, and it showed great promise.

After completing my base approach, I decided to implement multithreading to make the calculation even faster.(Multithreading means giving a program the ability to use more than one processor, which almost always results in an increase in speed). I had each thread doing the same task - calculating a stitch as fast as it could, and applying it to the video feed. With two threads, there was a great improvement over one thread. But with three threads, the improvement was negligible and did not justify adding additional complexity to the project.

Now that I knew two threads were ideal, I considered the best way to split tasks between them. I wanted to make sure that images always displayed in order, but I also wanted to minimize the time each thread spent waiting for the other. I realized that this meant that each thread should handle a different task.

I had one thread stitch images as fast as it could, based on the last available homography. The other thread calculated homography as fast as it could. This incorporated my idea of reusing homography for X frames, and also minimized the interaction between the two threads.

4.1 Results

Multithreading combined with information reuse offered a substantial speed improvement. The stitching method boasted 6.485 **Frames Per Second** and 1.904 **Homography Per Second**. This was 447% the FPS of traditional methods. The downside was that while the the image homography could properly warp the images, it was more difficult to align them. There was a noticeable offset between the frames, and because of the high frame rate it seems disjoint. As the camera shook, the two feeds felt very independent. If the videos were taken on a tripod to eliminate vertical movement, this method would work perfectly. However, this method would not work for a free camera video.

SINGLE THREADING SPEED VS MULTITHREADING SPEED

	TIME TO CALCULATE 100 FRAMES	TIME TO CALCULATE 28 HOMOGRAPHIES	FPS	HPS
SINGLE THREADING	68.95 seconds	19.31 seconds	1.450	1.450
MULTI-THREADING	15.42 seconds	14.70 seconds	6.485	1.904

PERCENT GAIN FPS = (6.485 -1.450)/1.450 = 347.24%

PERCENT GAIN HPS = (1.904-1.450)/1.450 = 31.31%

FPS = frames per second
HPS = homographies per second

5. COMBINED APPROACH?

At first glance, it seems possible to combine both approaches to further increase stitching speed. However, in practice it seems unlikely to succeed.

First of all, the FPS can not be boosted by using subimages in conjunction with multithreading. Subimages only improve the homography calculation time, because the final stitch always needs two complete images. Multithreading always offers the same FPS, and calculates homography independently. Therefore subimages would never be used at the stage that is responsible for FPS.

Additionally, although the homographies would be refreshed more often, the homographies would be less accurate. It does not seem like a worthwhile change to make; the gain in HPS is negligible in exchange for this loss of accuracy. (1.37 times faster, but with additional margin of error.)

The combined approach might have worked if the multithreaded approach was close to some kind of threshold. If this was the case, a small gain in homography speed could push it to the next level. Alas, the additional homography speed offered by the subimage approach still does not compensate for the alignment difficulties of the multithreaded approach.

Finally, the program would always need to occasionally calculate a complete stitch in order to remain accurate. Calculating the correct subimage boundaries requires sampling a complete image at some stage of the stitching pipeline. This means that some stitches would still have to be full image stitches, which further reduces the gain of sometimes using subimages. All in all, combining the approaches does not seem worthwhile.

6. DATA SUMMARY

SPEED OF EACH APPROACH

	FULL IMAGE	SUBIMAGE	MULTITHREADING	COMBINED (PROJECTED)
FPS	1.450	1.752	6.485	6.485
HPS	1.450	1.752	1.904	2.608

GAIN COMPARED TO FULL IMAGE

	SUBIMAGE	MULTITHREADING	COMBINED (PROJECTED)
FPS GAIN	.302	5.035	5.035
% FPS GAIN	**20.83%**	**347.24%**	**347.24%**
HPS GAIN	.302	0.454	1.158
% HPS GAIN	**20.83%**	**31.31%**	**79.86%**

Using subimages instead of full images does indeed improve the stitch speed, but the stitched image suffers a loss in accuracy. Because of the accuracy drop, using the subimage approach is not recommended even though it is 20.83% faster. The multithreaded approach shows a much more substantial gain in speed, but does not work unless the cameras are stationary. After creating both approaches, I was able to determine that combining these approaches would not be helpful to the stitching process.

7. CONCLUSION

This project contributed a new way of finding an overlapping region between two distinct images, and proved that both sub-image stitching and multithread stitching offer improvements in speed. In the future it would be possible to further improve the accuracy of the stitches by implementing blending algorithms, fully automating seam detection, and implementing feature point tracking to dynamically adjust the image translations.

If computing hardware continues to improve, it is possible that one day we will get to a threshold where these approaches can finally nudge the calculation the rest of the way.

FPS = frames per second
HPS = homographies per second

8. REFERENCES

[1] Golla, Ramsri. "Panorama – Image Stitching in OpenCV." *Ramsrigoutham*. 11 Nov. 2012. Web. 05 May 2016.
[2] "Introduction to SURF (Speeded-Up Robust Features)." *Introduction to SURF (Speeded-Up Robust Features) — OpenCV 3.0.0-dev Documentation*. 10 Nov. 2014. Web. 05 May 2016.
[3] Pele, Ofir. "SIFT - The Scale Invariant Feature Transform." *Freie Universitat Berlin*. Web. 5 May 2016.
[4] "High Level Functionality." *High Level Functionality — OpenCV 2.4.13.0 Documentation*. 5 May 2016. Web. 05 May 2016.
[5] "Stitching Pipeline." *Stitching Pipeline — OpenCV 2.4.13.0 Documentation*. 5 May 2016. Web. 05 May 2016.
[10] "Features and Image Matching." *Computer Science and Engineering, University of Washington*. Web. 5 May 2016. <https://courses.cs.washington.edu/courses/cse455/09wi/Lects/lect6.pdf>.
[11] Verma, Chaman Singh, and Mon-Ju. "Panoramic Image Mosaic." *Imagemosaic*. 5 May 2016. Web. 05 May 2016.
[12] "FLANN - Fast Library for Approximate Nearest Neighbors : FLANN - FLANN Browse." *FLANN - Fast Library for Approximate Nearest Neighbors : FLANN - FLANN Browse*. 5 May 2016. Web. 05 May 2016.
[13] Kriegman, David. *Homography Estimation*. UC San Diego, 5 May 2016. Web. 5 May 2016.
[14] "The Homography Transformation." *The Homography Transformation*. Corrmap. Web. 05 May 2016.
[15] Gizmo, Richards. "Best Free Digital Image Stitcher." *Gizmo's Freeware*. 28 Apr. 2015. Web. 05 May 2016.
[16] "PTGui." *Photo Stitching Software 360 Degree Panorama Image Software*. Web. 05 May 2016.
[17] "360 Video Made Easy | VideoStitch - VR Video Solutions." *360 VR Video Software VideoStitch*. Web. 05 May 2016.
[18] Appello, Daniel, and Danielle Erickson. "Multi-Cam Video Stitching: Panoramic Video Option During a Skype Call." Thesis. Dickinson College, 1. *Multi-Cam Video Stitching: Panoramic Video Option During a Skype Call*. Dickinson College, 9 May 2014. Web. 5 May 2016.
[19] Yeager, Luke. "StitcHD Demo 1." *YouTube*. YouTube, 08 May 2012. Web. 05 May 2016.
[20] Varavva, Ievgen. "Simple Video Image Stitching with Opencv." *YouTube*. YouTube, 03 May 2011. Web. 05 May 2016.
[21]"Blending." *Carnegie Melon Graphics*. Web. 5 May 2016. <http://graphics.cs.cmu.edu/courses/15-463/2010_spring/Lectures/blending.pdf>.